Quantitative Analysis of Floating Point Arithmetic on FPGA Based Custom Computing Machines

Nabeel Shirazi, Al Walters, and Peter Athanas

Virginia Polytechnic Institute and State University
Department of Electrical Engineering
Blacksburg, Virginia 24061-0111

shirazi@pequod.ee.vt.edu

Abstract

Many algorithms rely on floating point arithmetic for the dynamic range of representations and require millions of calculations per second. Such computationally intensive algorithms are candidates for acceleration using custom computing machines (CCMs) being tailored for the application. Unfortunately, floating point operators require excessive area (or time) for conventional implementations. Instead, custom formats, derived for individual applications, are feasible on CCMs, and can be implemented on a fraction of a single FPGA. Using higher-level languages, like VHDL, facilitates the development of custom operators without significantly impacting operator performance or area. Properties, including area consumption and speed of working arithmetic operator units used in real-time applications, are discussed.

1.0 Introduction

Until recently, any meaningful floating point arithmetic has been virtually impossible to implement on FPGA based systems due to the limited density and speed of older FPGAs. In addition, mapping difficulties occurred due to the inherent complexity of floating point arithmetic. With the introduction of high level languages such as VHDL, rapid prototyping of floating point units has become possible. Elaborate simulation and synthesis tools at a higher design level aid the designer for a more controllable and maintainable product. Although low level design specifications were alternately possible, the strategy used in the work presented here was to specify every aspect of the design in VHDL and rely on automated synthesis to generate the FPGA mapping.

Image and digital signal processing applications typically require high calculation throughput [2,6]. The arithmetic operators presented here were implemented for real-time signal processing on the Splash-2 CCM, which include a 2-D fast Fourier transform (FFT) and a systolic array implementation of a FIR filter. Such signal processing techniques necessitate a large dynamic range of numbers. The use of floating point helps to alleviate the underflow and overflow problems often seen in fixed point formats. An advantage of using a CCM for floating point implementation is the ability to customize the format and algorithm data flow to suit the application's needs.

This paper examines the implementations of various arithmetic operators using two floating point formats similar to the IEEE 754 standard [5]. Eighteen and sixteen bit floating point adders/subtracters, multipliers, and dividers have been synthesized for Xilinx 4010 FPGAs [8]. The floating formats used are discussed in Section 2. Sections 3, 4, and 5 present the algorithms, implementations, and optimizations used for the different operators. Finally a summary, in terms of size and speed, of the different floating point units is given Section 6.

2.0 Floating Point Format Representation

The format which was used is similar to the IEEE 754 standard used to store floating point numbers. For comparison purposes, single precision floating point uses the 32 bit IEEE 754 format shown in Figure 1.

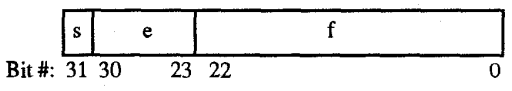

Figure 1: 32 Bit Floating Point Format.

0-8186-7086-X/95 $04.00 © 1995 IEEE

The floating point value (v) is computed by:

$$v = -1^s\, 2^{(e - 127)}(1.f)$$

In Figure 1, the sign field, s, is bit 31 and is used to specify the sign of the number. Bits 30 down to 23 are the exponent field. This 8 bit quantity is a signed number represented by using a bias of 127. Bits 22 down to 0 are used to store the binary representation of the floating point number. The leading one in the mantissa, 1.f, does not appear in the representation, therefore the leading one is implicit. For example, -3.625 (dec) or -11.101 (binary) is stored in the following way:

$$v = -1^1\, 2^{(128-127)}1.1101$$

where:

s = 1, e = 128 (dec) 80 (hex), and f = 680000 (hex).

Therefore -3.625 is stored as: C0680000 (hex).

The 18-bit floating point format was developed, in the same manner, for the 2-D FFT application[6]. The format was chosen to accommodate two specific requirements: (1) the dynamic range of the format needed to be quite large in order to represent very large and small, positive and negative real numbers accurately, and (2) the data path width into one of the Xilinx 4010 processors of Splash-2 is 36 bits wide and two operands were needed to be input on every clock cycle. Based on these requirements the format in Figure 2 was used.

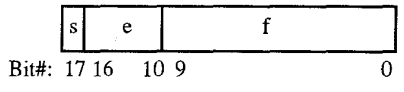

Figure 2: 18 Bit Floating Point Format.

The 18 bit floating point value (v) is computed by:

$$v = -1^s\, 2^{(e - 63)}(1.f)$$

The range of real numbers that this format can represent is $\pm 3.6875 \times 10^{19}$ to $\pm 1.626 \times 10^{-19}$.

The second floating point format investigated was a 16-bit representation used by the FIR filter application [7]. Like the FFT application, since multiple arithmetic operations needed to be done on a single chip, we chose a 16-bit format for two reasons: (1) local, 16-bit wide memories were used in pipelined calculations allowing single read cycles only, and (2) more logic was necessary to implement the FIR taps in addition to the two arithmetic units. which do complex number operations. The format was designed as a compromise between data width and a large enough dynamic number range. The 16-bit format is shown in Figure 3.

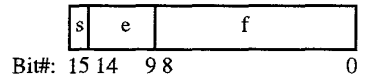

Figure 3: 16 Bit Floating Point Format.

The 16 bit floating point value (v) is computed by:

$$v = -1^s\, 2^{(e - 31)}(1.f)$$

The range of real numbers that this 16 bit format can represent is $\pm 8.5815 \times 10^9$ to $\pm 6.985 \times 10^{-10}$.

3.0 Floating-Point Addition and Subtraction

The aim in developing a floating point adder/subtracter routine was to pipeline the unit in order to produce a result every clock cycle. By pipelining the adder, the speed increased, however, the area increased as well. Different coding structures were tried in the VHDL code used to program the Xilinx chips in order to minimize size.

3.1 Algorithm

The floating-point addition and subtraction algorithm studied here is similar to what is done in most traditional processors, however, the computation is performed in three stages and is presented in this section. The notation s_i, e_i and f_i are used to represent the sign, exponent and mantissa fields of the floating point number, v_i. A block diagram of the three-stage adder is shown in Figure 4. The computations required for each stage are as follows:

Stage 1:

- If the absolute value of v_1 is less than the absolute value of v_2 then swap v_1 and v_2. The absolute value is checked by comparing the exponent and mantissa of each value.
- Subtract e_2 from e_1 in order to calculate the number of positions to shift f_2 to the right so that the decimal points are aligned before addition or subtraction in Stage 2.

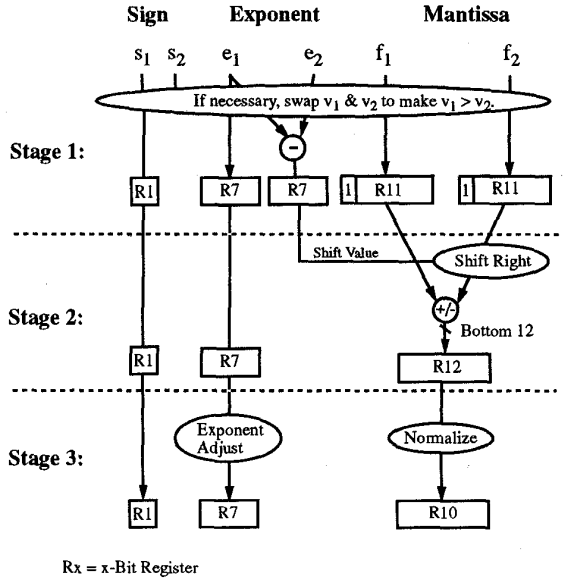

Figure 4: Three stage 18-bit Floating Point Adder.

Stage 2:

- Shift $1.f_2$ to the right (e_2 - e_1) places calculated in the previous stage.
- Add $1.f_1$ to $1.f_2$ if s_1 equals s_2, else subtract $1.f_2$ from $1.f_1$.
- Set the sign and the exponent of the final result, v_3, to the sign and the exponent of the greater value v_1.

Stage 3:

- Normalization of f_3 is done by shifting it to the left until the high order bit is a one.
- Adjusting exponent of the result, e_3, is done by subtracting it by the number of positions that f_3 was shifted left.

3.2 Eighteen Bit Floating Point Addition Example

To demonstrate an 18 bit, 3 stage floating point adder we add $v_1 + v_2 = v_3$, where $v_1 = 24.046875$ and $v_2 = -25.40625$. Therefore v_3 should equal -1.359375.

	Decimal	Binary	18 Bit Format
v_1	24.046875	1.1000000011×2^4	0 1000011 1000000011
v_2	-25.40625	1.1001011010×2^4	1 1000011 1001011010

Therefore: $s_1 = 0$ $e_1 = 1000011$ $1.f_1 = 1.1000000011$
$s_2 = 1$ $e_2 = 1000011$ $1.f_2 = 1.1001011010$

Stage 1:

- Swap v_1 and v_2 since $e_1 = e_2$ and $f_2 > f_1$
  Now: $s_1 = 1$ $e_1 = 1000011$ $1.f_1 = 1.1001011010$
  $s_2 = 0$ $e_2 = 1000011$ $1.f_2 = 1.1000000011$
- Since $e_1 - e_2 = 0$, $1.f_2$ does not need to be shifted in the next stage.

Stage 2:

- Since s_1 does not equal s_2, $1.f_3 = 1.f_1 - 1.f_2$.
- Also, $s_3 = f_1$ and $e_3 = e_1$ since they are the sign and exponent of the greater value.
  After stage 2: $s_3 = 1$ $e_3 = 1000011$ $1.f_3 = 0.0001010111$

Stage 3:

- Normalize f_3 by shifting it 5 places to the left.
- Adjust the Exponent, e_3, by subtracting 5 from it.
  After final stage: $s_3 = 1$ $e_3 = 0111111$ $1.f_3 = 1.0101110000$

The result, v_3, after addition is shown as follows:

	Decimal	Binary	18 Bit Format
v_3	-1.359375	1.010111×2^0	0 0111111 0101110000

3.3 Optimization

The circuits produced by contemporary VHDL synthesis tools are, unfortunately, highly sensitive to the manner in which the original behavioral or structural description is expressed. When designing the floating point adder/subtracter, using different VHDL constructs to describe the same behavior resulted in a faster and smaller design.

The parts of the adder which caused the bottleneck were the exponent subtracter, the mantissa adder/subtracter and the normalization unit. An 8-bit and a 16-bit Xilinx hard-macro adder/subtractor[8] was used in place of VHDL code written for the exponent and mantissa computation. This increased the overall speed of the design even though a smaller 12-bit adder/subtracter was replaced with a 16-bit adder/subtracter hard macro. The first cut at the normalization unit resulted in a very slow and large design. VHDL *for* loops were used for the shift left and for the code that finds the most significant bit during normalization. In order to decrease the size and increase the speed of the design, the *for* loops were unrolled and *if* statements used instead.

The first method used for shifting the mantissa of the second operand, f_2, a variable number of places was originally coded in VHDL the following way:

```
-- Shift f2 right ediff places
  e_diff_var := e_diff;
  f2_var := f2(10 downto 0);

  for i in 1 to 11 loop
    if (e_diff_var > zero_8) then
      f2_var(9 downto 0) := f2_var(10 downto 1);
      f2_var(10) := '0';
      e_diff_var := e_diff_var - 1;
    end if;
  end loop;

  f2_result(10 downto 0) <= f2_var;
```

The second method used *if* statements to check each individual bit of the shift value and shift f_2 accordingly.

```
  -- Shift f2 right ediff places
  if ((e_diff(7) = '1') or (e_diff(6) = '1') or
(e_diff(5) = '1') or (e_diff(4) = '1')) then
      e_diff_var(3 downto 0) := "1111";
  else
      e_diff_var(3 downto 0) := e_diff(3 downto 0);
  end if;

  -- Sequential Code for shifting f2_var
  f2_var := f2(10 downto 0);
  if (e_diff_var(0) = '1') then
      f2_var(9 downto 0) := f2_var(10 downto 1);
      f2_var(10) := '0';
  end if;
  if (e_diff_var(1) = '1') then
      f2_var(8 downto 0) := f2_var(10 downto 2);
      f2_var(10 downto 9) := "00";
  end if;
  if (e_diff_var(2) = '1') then
      f2_var(6 downto 0) := f2_var(10 downto 4);
      f2_var(10 downto 7) := "0000";
  end if;
  if (e_diff_var(3) = '1') then
      f2_var(2 downto 0) := f2_var(10 downto 8);
      f2_var(10 downto 3) := "00000000";
  end if;

  f2_result(10 downto 0) <= f2_var;
```

The result of using the second method is shown in Table 1. The variable 11-bit shifter became two times smaller and three times faster.

	Method 1	Method 2	Advantage
FG Function Generators (used/available)	85/800 10%	44/800 5%	2x smaller
Flip Flops	6%	6%	same
Speed	6.5 MHz	19.0 MHz	2.9x faster

TABLE 1. Optimizations for Variable 11-Bit Barrel Shifter.

The code used to normalize the result after addition or subtraction of f_1 and f_2 was also initially written using *for* loops in VHDL.

```
-- Shift f_result left until msb = 1
msb := f(10);
f_result_var := f;
e_result_var := e;

for i in 1 to 11 loop
 if (msb = '0') then
  f_result_var(10 downto 1) := f_result_var(9 downto 0);
  f_result_var(0) := '0';
  e_result_var := e_result_var - 1;
  msb := f_result_var(10);
 end if;
end loop;

f_result <= f_result_var(9 downto 0);
e_result <= e_result_var;
```

The second method calculates the number of places to shift the mantissa to the left in order to position the most significant bit (msb) in the high order location. A series of *if* statements are used to check all possible bit position locations for the msb in order to calculate the shift value. After the shift value is calculated, a procedure similar to the second method for performing a variable shift to the right is used to shift the un-normalized value the correct number of positions to the left in order to normalize it. Due to the size of the VHDL source code it is not listed here for this method.

By using the second method, the normalization unit became 2.9 times smaller and 2.6 times faster. A summary of the result of optimizing the normalization unit is shown in Table 2.

	Method 1	Method 2	Advantage
FG Function Generators	167/800 20%	58/800 7%	2.9x smaller
Flip Flops	6%	6%	same
Speed	5.1 MHz	13.4MHz	2.6x faster

TABLE 2. Optimizations for Normalization Unit.

The overall size and speed of the 16 and 18-bit floating point adders are given in Section 6.

4.0 Floating Point Multiplication

Floating point multiplication is much like integer multiplication. Because floating-point numbers are stored in sign-magnitude form, the multiplier needs only to deal with unsigned integer numbers and normalization. Like the architecture of the floating point adder, the floating point multiplier unit is a three stage pipeline that produces a result on every clock cycle. The bottleneck of this design was the integer multiplier. Four different methods were used to optimize the integer multiplier in order to meet speed and size requirements.

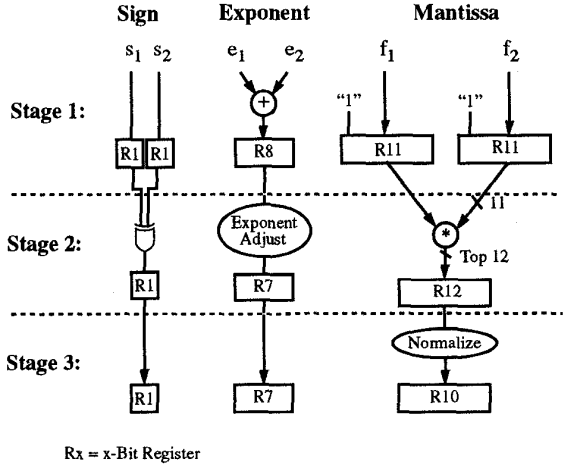

Figure 5: Three stage 18 bit Floating Point Multiplier.

4.1 Algorithm

The block diagram for the three stage 18 bit floating point multiplier is shown in Figure 5. The algorithm for each stage is as follows:

Stage 1:

- The exponents, e_1 and e_2 are added and the result along with the carry bit is stored in an 8-bit register. If the addition of two negative exponents results in a value smaller than the minimum exponent that can be represented, i.e. -63, underflow occurs. In this case the floating point number is set to zero. If overflow occurs, the result is set to the maximum number the format can represent.
- If the floating point number is not zero, the implied one is concatenated to the left side of the f_1 and f_2 terms.
- The sign bits are only registered in this stage.

Stage 2:

- Integer multiplication of the two 11-bit quantities, $1.f_2$ and $1.f_1$, is performed. The top 12 bits of the 22-bit result is stored in a register.
- The exponent is adjusted depending on the high order bit of the multiplication result.
- The sign bits of the two operands are compared. If they are equal to each other the result is assigned a positive sign, and if they differ the result is negative.

Stage 3:

- Normalization of the resulting mantissa is performed.
- The resulting sign, exponent and mantissa fields placed into an 18-bit floating point word.

4.2 Optimization

Four different methods were used to optimize the 11-bit integer multiplier. The first method used the integer multiply available in the Synopsys 3.0a VHDL compiler. The second method was a simple array multiplier composed of ten 11-bit carry-save adders [3]. The last two methods involved pipelining the multiplier in order to increase the speed of the design. The multiplication of the two 11-bit quantities were broken up and multiplied in the following way:

```
      X6 X5
   *  Y6 Y5
      X5Y5
     X6Y5
    X5Y6
+ X6Y6
22 Bit Result
```

In the third method, the first stage of the multiplier was the multiplication of the four terms X5Y5, X6Y5, X5Y6, and X6 Y6. The second stage involved adding the results of the four multiplications. In method 4, two stages were used to sum the multiplication terms.

The results of the four methods are summarized in Table 3. The advantage in terms of the number of times faster and the number of times larger than Method 1 is shown.

	Method 1	Method 2	Method 3	Method 4
FG Function Generators	35%	31%	45%	47%
Stages	1	1	2	3
Speed	4.9 MHz	3.7 MHz	6.2 MHz	9.4 MHz
Area Advantage	1.0	0.90	1.29	1.34
Speed Advantage	1.0	0.75	1.24	1.92

TABLE 3. Results from Four Methods Used to Optimize an Integer 11-Bit Multiplier.

Method 1 was used in the floating point multiplier unit since the size of the unit was too large using methods 3 or 4 to allow an additional floating point unit in the same chip. The overall size and speed of the 16 and 18-bit floating point multipliers are given in Section 6, Summary and Conclusions.

5.0 Floating Point Division

A floating-point division technique is presented here which utilizes the pipelined multiplier discussed earlier. Division can be done by using the reciprocal of the divisor value so that the equation for division becomes a multiplication of (A x ($1/B$) = Q). Independent operations on the different floating point fields enable the design to be pipelined easily.

5.1 Algorithm

The reciprocal of a floating point value can be accomplished in two steps: (1) reciprocate the mantissa value, and (2) negate the power of the base value. Since the floating point representation already has its fields segregated, the task becomes trivial for a processing element which is complemented by a memory bank of at least 2^n x n bits, where n is the size of the mantissa's normalized binary representation. Local memories to the processing elements store the reciprocal of each bit combination of the mantissa.

In order to pipeline the design, three steps prior to the multiplication are necessary: (1) extract the mantissa from the input as the memory address and negate the exponent, (2) provide a delay until the memory data is valid, and (3) insert the new mantissa. The data word created during Stage 3 is passed to the multiplier. The second stage of the pipeline depends on the system being used which could result in longer delays before the data is made available from memory. In this case, a Splash 2 implementation was used which is shown in Figure 6. Memory reads require a single cycle after the address is presented before the data can be acquired from the memory data buffer.

The k_1 value negates the exponent, which still retains the embedded excess value. Note that since the reciprocal of the given mantissa value will be less than or equal 1.0, normalization for the mantissa has to be done,

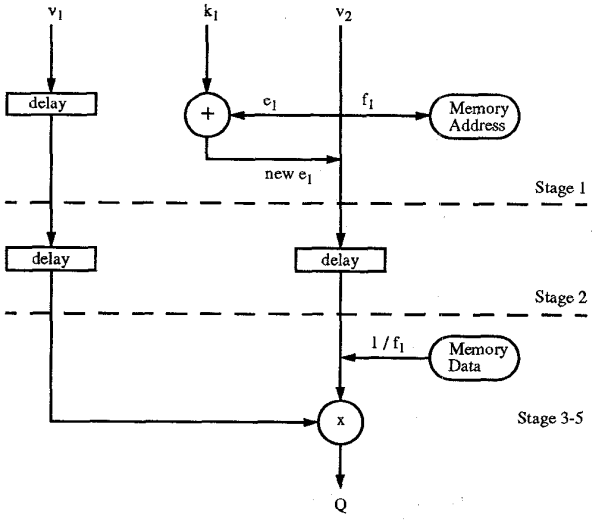

Figure 6: Three stage 18 bit Floating Point Divider.

although a special case for 1.0 has to be made. The normalization process is done automatically with k_1. Once the addition is done, the result becomes the new exponent passed onto Stage 2. The mantissa in Stage 1 directly goes to the memory address buffer to obtain the new mantissa, but the old mantissa continues into Stage 2 and is replaced in Stage 3. Stage 2 of the pipeline waits for the data to become available from the memory. This occurs at Stage 3. The new mantissa is inserted into the final operand to be passed to the multiplier. Although three pipeline stages are shown here, additional stages occur due to the pipelined multiplier to make a total of five stages.

6.0 Summary and Conclusions

The aim in designing the floating point units was to pipeline each unit a sufficient number of times in order to maximize speed and to minimize area It is important to note that once the pipeline is full, a result is output every clock cycle. A summary of the resulting size and speed of the 16 bit and 18 bit floating point units is given in Tables 4 and 5 respectively.

The Synopsys Version 3.0a VHDL compiler was used along with the Xilinx 5.0 tools to compile the VHDL description of the floating point arithmetic units. The Xilinx timing tool, *xdelay*, was used to estimate the speed of the designs.

	Adder/ Subtracter	Multiplier	Divider
FG Function Generators	26 %	36%	38%
Flip Flops	13 %	13%	32%
Stages	3	3	5
Speed	9.3 MHz	6.0 MHz	5.9 MHz

TABLE 4. Summary of 16 bit Floating Point Units.

To implement single precision floating point arithmetic units on the Splash-2 architecture, the size of the floating point arithmetic units would increase between 2 to 4 times over the 18 bit format. A multiply unit would require two Xilinx 4010 chips and an adder/subtracter unit

	Adder/ Subtracter	Multiplier	Divider
FG Function Generators	28%	44%	46%
Flip Flops	14%	14%	34%
Stages	3	3	5
Speed	8.6 MHz	4.9 MHz	4.7 MHz
Tested Speed	10 MHz	10 MHz	10 MHz

TABLE 5. Summary of 18 bit Floating Point Units.

broken up into four 12-bit multipliers, allocating two per chip[4]. We found that a 16x16 bit multiplier was the largest parallel integer multiplier that could fit into a Xilinx 4010 chip. When synthesized, this multiplier used 75% of the chip area

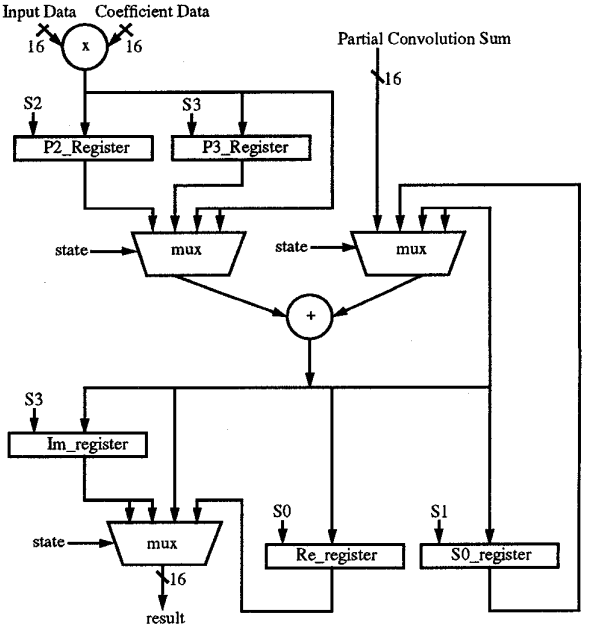

Figure 7: The diagram shows a single Splash-2 PE design for an FIR tap to accomplish complex multiplication. The architecture can achieve two floating-point calculations per clock cycle.

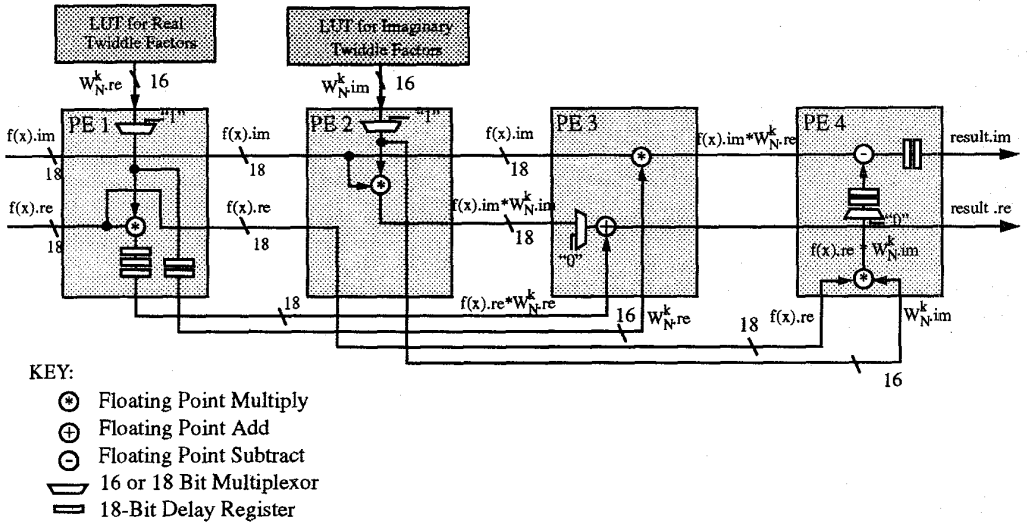

Figure 8: A block diagram of a four PE Splash-2 design for a complex floating point multiplier used in a FFT butterfly operation. Six floating operations are calculated every clock cycle at 10 MHz.

Each of the floating point arithmetic units has been incorporated into two applications: a 2-D FFT [6] and a FIR filter [7]. The FFT application operates at 10 MHz and the results of the transform are stored in memory on the Splash-2 array board. These results were checked by doing the same transform on a SPARC workstation An FIR tap design using a floating point adder and multiplier unit is shown in Figure 7. The complex floating point multiplier used in the 2-D FFT butterfly calculation is shown in Figure 8.

Acknowledgments

We wish to express our gratitude to Dr. J. T. McHenry and Dr. D. Buell. We would also like to thank Professor J. A. DeGroat of The Ohio State University for technical advice.

This research has been supported in part by the National Science Foundation (NSF) under grant MIP-9308390.

References

[1] J.M. Arnold, D.A. Buell and E.G. Davis, "Splash 2," *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 316-322, June 1992.

[2] J.A. Eldon and C. Robertson, "A Floating Point Format for Signal Processing," *Proceedings IEEE International Conference on Acoustics, Speech, and Signal Processing*, pp. 717-720, 1982.

[3] K. Eshraghian and N.H.E. Weste, *Principles of CMOS VLSI Design, A Systems Perspective*, 2nd Edition, Addison-Wesley Publishing Company, 1993.

[4] B. Fagin and C. Renard, "Field Programmable Gate Arrays and Floating Point Arithmetic," *IEEE Transactions on VLSI*, Vol. 2, No. 3, pp. 365-367, September 1994.

[5] IEEE Task P754, "A Proposed Standard for Binary Floating-Point Arithmetic," IEEE Computer, Vol. 14, No. 12, pp. 51-62, March 1981.

[6] N. Shirazi, *Implementation of a 2-D Fast Fourier Transform on an FPGA Based Computing Platform*, VPI&SU Masters Thesis in progress.

[7] A. Walters, *An Indoor Wireless Communications Channel Model Implementation on a Custom Computing Platform*, VPI&SU Master Thesis in progress.

[8] Xilinx, Inc., *The Programmable Logic Data Book*, San Jose, California, 1993.